Final Report project AI Applications: AI4IM (simulated data Y2 labels)

Student: Alfredo Vargas

R-number: r0835034

Problem description

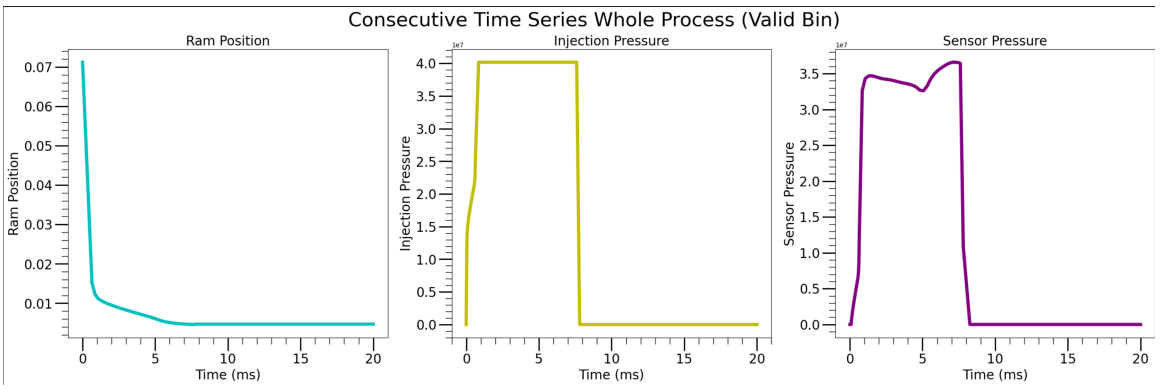
- Simulated Data of bin productions using Injection Moulding (Datapoints generated using Matlab)

Dataset

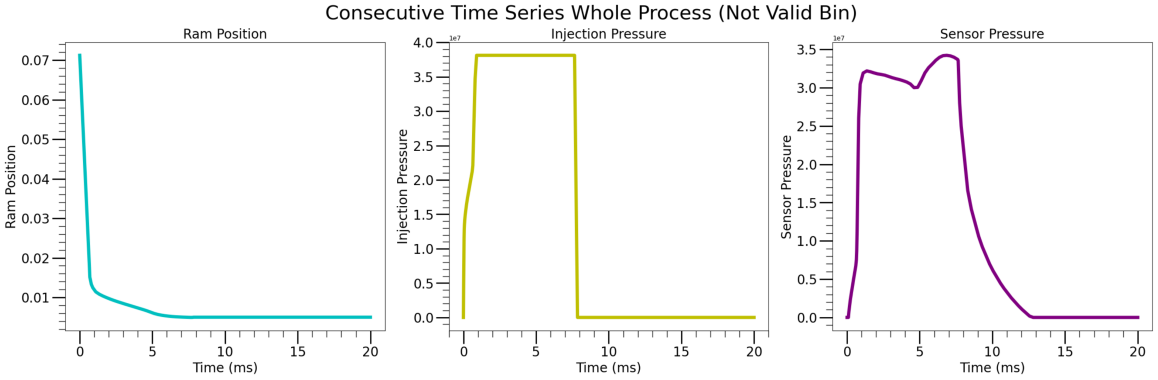
- 1542 datapoints
- The datapoints contains three time series:
  i. Ram position vs time
  ii. Injection Pressure vs time
  iii. Sensor Pressure vs time

Data Exploration

- A valid bin multivariate time series process looks as follows:



- A not valid bin multivariate time series process looks as follows:



- Although sometimes we see a differences between these two random selected points for valid and not valid. We cannot conclude that there is deterministic method to determine when a bin will have a defect or not. However, we see that there could be enough information to make statistical analysis, therefore ML methods can be suitable.

Labels:

- $Y_2 \in \mathbb{B}^{1542\times 1}$, where $B = \{0, 1\}$ with 1 representing a valid bin and 0 a not valid bin.
- Description:

Feature	Dimension	Data Type
ramposition	1542	float
ramposition_time	1542	float
injection_pressure	1542	float
injection_pressure_time	1542	float
sensor_pressure	1542	float
sensor_pressure_time	1542	float

- Imbalanced dataset:
  - The number of valid bins is 1080 which correspond to the 70.04 %
  - The number of not valid bins is 462 which correspond to the 29.96 %

Goal:

- Selecting the metric:
  - It is important to note that in a time series problem the mean accuracy

cannot be considered as a good indicator for the performance. The reason for this is that the datapoins are time correlated and are therefore not independent from each other. However, after feature engineering which allows to represent the time series problem as a classification problem with tabular data, then we at this point can consider the mean accuracy as good indicator for the performance of the algorithm **ONLY** when we have a balanced dataset!. We address this imbalance issue by incrementing the minority class and using the **f1 score** to estimate better the true positives, true negatives, false positives and false negatives.

- Get at least a performance of 80% for the f1-score for both the majority class (valid bins) and minority class (not valid bins).

- From the economical point of view of a molding injection process, not detecting the faulty pieces (**false positives**) is the one that have the most negatively impact, *both economically and to the environment*:

    - Economically, because we are not able to efficiently use the material and resources that require to follow the quality standard of the bins. This is a condition before one is able to commercialize any product.
    - Environmentally, because we waste `plastic` material whose usage we are already trying to minimize.

Data preprocessing & Feature engineering

Feature engineering with `Helper.py`:

1. `series2features` function from `Helper.py` file, for each time-series generates 22 new engineered features.
2. After that concatenated features into one dataset matrix with dimension 66 features plus 1 label Y_2.

Feature engineering with `tsfresh`:

1. First trim the values **before** implementing feature engineering.
2. Concatenate the time series before implementing feature engineering with `tfresh` to incorporate the effects of a multivariate time series problem.
3. Select the most relevant features by specifying a `p-value` (This parameter was optimized!)

Data cleaning steps:

- Some small features are trimmed whenever their values are smaller than the trim value limit $\epsilon = 0.00001$ (trim value)
- Some features have a constant value which can be dropped as they not contribute when one deals with some ML methods such a decision trees. However, we could keep it for its use when using other ML methods.
- To drop the features we use:

```
full_data_df.dropna(axis=1).describe()  # we drop along the columns axis=
```

Pre-processing steps:

- Normalization done as follows: (standardization)

```
cols_to_norm = [i for i in range(0, 66)]  # we exclude the labels
full_data_df[cols_to_norm] = full_data_df[cols_to_norm].apply(lambda x: (
full_data_df.head()
```

- All of the claning steps were implemented using pipelines within the pandas framework (see notebooks).

Data splits, tried the following:

1. Split 80% for train and 20% for test. A sort of training "cross-validation" can be considered intrinsically incorporated when using Random Forest with a number of estimator larger than one (the default is 100). To split the data we used:

```
from sklearn.model_selection import train_test_split
```

2. $p - value$ tuning to determine how many features must be kept in order to obtain the best results (**f1 scores**).
3. All data splits used the `random_state=0`. So one can easily reproduce and compare the performance between algorithms.

Selecting the Machine Learning Model

Methods

- Because were dealing with a classification problem but not a regression, we need

to consider models that allows to classify.

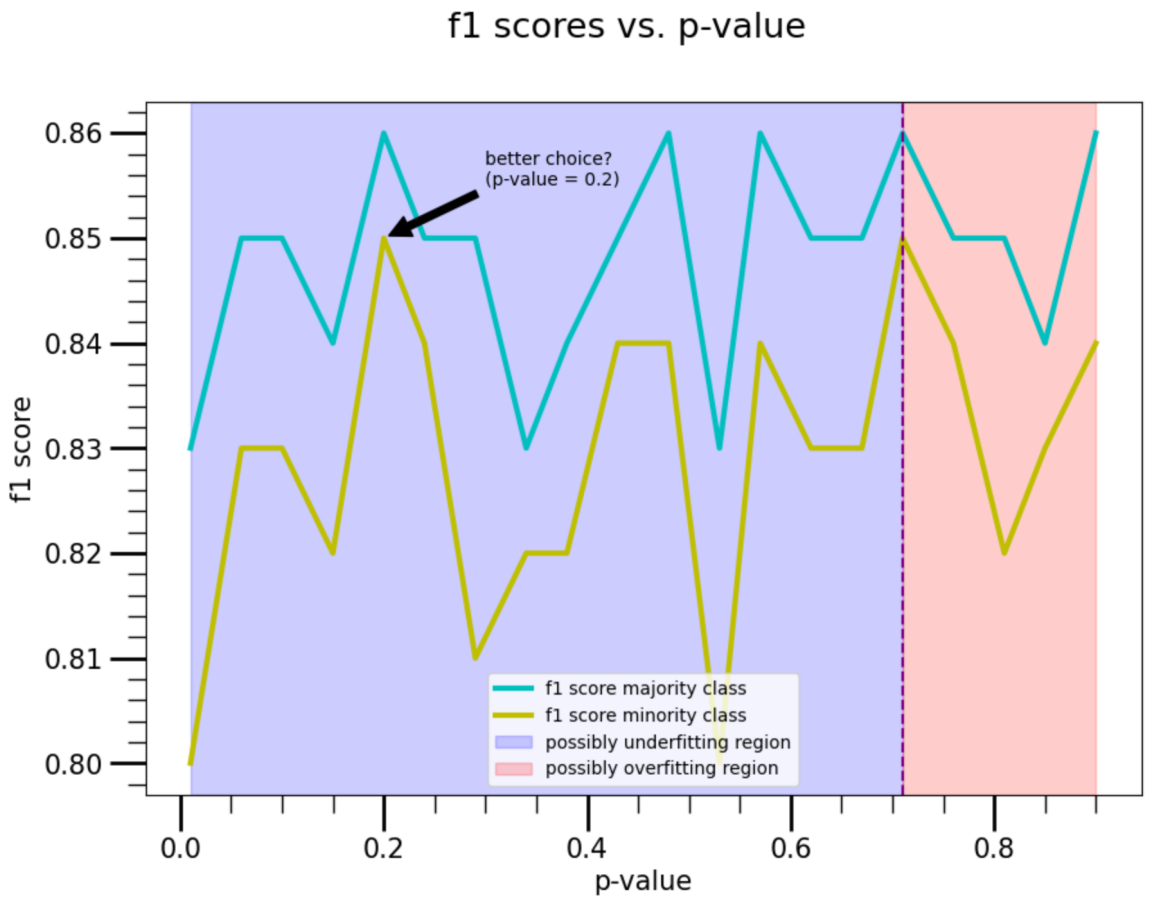
- Let's start with:
  i. Decision trees & Ensembles (specifically RandomForest)
  ii. kNN
  iii. Neural Networks

1. Random Forest Classifier

   1.1 With Helper

   - f1 score of minority and majority classes is 0.80

   1.2 With Tsfresh



   - f1 score of minority and majority classes is 0.86 and 0.87

2. kNN

   - Optimal number of neighbors: 29
   - f1 score of minority and majority classes is 0.72 and 0.80

3. Multi-layer Perceptron Classifier

   3.1 Number of features filtered:

   - The datasets with pv=0.01 had f1 score for the minority and majority clases

of: 0.68 and 0.47
    - The datasets with pv=0.06 had f1 score for the minority and majority clases of: 0.69 and 0.78
    - The datasets with pv=0.1 had f1 score for the minority and majority clases of: 0.69 and 0.68
    - The datasets with pv=0.15 had f1 score for the minority and majority clases of: 0.70 and 0.75
    - The datasets with pv=0.2 had f1 score for the minority and majority clases of: – and – (crashes whenever we reproduce this step)
    - The datasets with pv=0.24 had f1 score for the minority and majority clases of: 0.70 and 0.78

- Higher values of pv are still needed to be explored. However the f1 score not seems to improve that much for the minority class, so we will pick pv = 0.06 as our starting dataset.

Results

Performance Table

Metric \ Model	Random forest from Helper FE	Random forest from tsfresh FE (pv=0.68)	KNN from tsfresh	MLPC from tsfresh
Precision for not valid class	0.79	0.84	0.74	0.78
Precision for the valid class	0.81	0.89	0.81	0.72
Recall valid for not valid class	0.87	0.81	0.64	0.61
Recall for the valid class	0.79	0.86	0.88	0.85
F1-score for not valid	0.80	0.86	0.72	0.69

Metric \ Model	Random forest from Helper FE	Random forest from tsfresh FE (pv=0.68)	KNN from tsfresh	MLPC from tsfresh
class				
F1-score for the valid class	0.80	0.87	0.80	0.78

Conclusions

Data Exploration and Pre Analysis

- From the figures, we observe that potentially one has the opportunity to differentiate the bin classes valid and not valid by using the `Helper.py` module and its corresponding feature engineered dataset.
- From the new engineered dataset we observer that we will require some preprocessing. More specifically we will create in what follows a pipeline that will perform the following:
  - Standarization
  - Removal of NaN values
  - Balancing of the dataset (here we will use the `imblearn.over_sampling` and from it the `SMOTE` function):

```
from imblearn.over_sampling import SMOTE
```

- We can clearly identified two regions when feature engineering one related to the under fitting region when `p-values` when using the `tfresh` . One region corresponds to the under-fitting region ($p-value<0.48$), meaning we have less features (52% or more are considered rare features and therefore ignored). The other region corresponds to the over-fitting region with $p-value>0.48$ (52% or less are considered rare features and therefore ignored.)

Avoiding Overfitting

- We tried to control the number of engineered features used to reduce the complexity of the model and therefore reducing overfiting. The `p-value` that generated the best score with a lower number of features should be used for this case.

- Cross-validation can also help with overfitting. This was implemented whenever we performed a GridSearch, the default for this particular case of `cv=None` implies $5-fold$ cross-validation.

Explainability

- During the feature engineering process `p-value` tuning allows to detect high correlated features in a dataset. If one selects purposely a dataset with the lower p-value and higher performance, one can more easily extract insights that could become relevant for the manufacturing process of the bins in question or the injection moulding in question.

- The dataset with `p-value=0.01` has the lowest amount of features and quite high precision (above 80). Those features include mostly some Fourier transform coefficients, among others parameters. Meaning that highly localized points in the configuration space of the features are the most relevant to make the distinction between a valid and not valid classes. Tracking those parameters could also give insight of what features are the most important when engineering the studied bins.

Reproducibility

- Whenever we split the data between training and test we used a `random_state=0` .
- For the `RandomForestClassifier` and `MLPClassifier` we used `random_state=123` , whenever bootstrapping was involved in **Random Forest** or for the generation of the initial weights and bias for the **Multi-layer Perceptron Classifier**.
- The balancing of the data using the `SMOTE` library also required some `random_state` . We avoided this issue by saving the datasets after balancing them and the using those for training.
- To create the `conda` environment used to run the present notebooks, use:

```
conda env create --file ai4im.yml
```

- The github repo with all notebooks and relevant files are in the following repo:
  - https://github.com/Alfredo-Vargas/ai4im

Bibliography

- Feature Engineering using `tsfresh`:
  - Pycon:DE 2017: “Get rich or die … overfiting”:
    - https://www.youtube.com/watch?v=Fm8zcOMJ-9E&ab_channel=PyConDE
  - tsfresh documentation:
    - https://tsfresh.readthedocs.io/en/latest/
- Oversampling technique `SMOTE` original paper:
  - https://arxiv.org/abs/1106.1813